

UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Programowanie Robotów Mobilnych

Robot Operating System

Rafał Szczepański
e-mail: szczepi (at) umk.pl
www.umk.pl/~szczepi

07.03.2023



UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Plan prezentacji

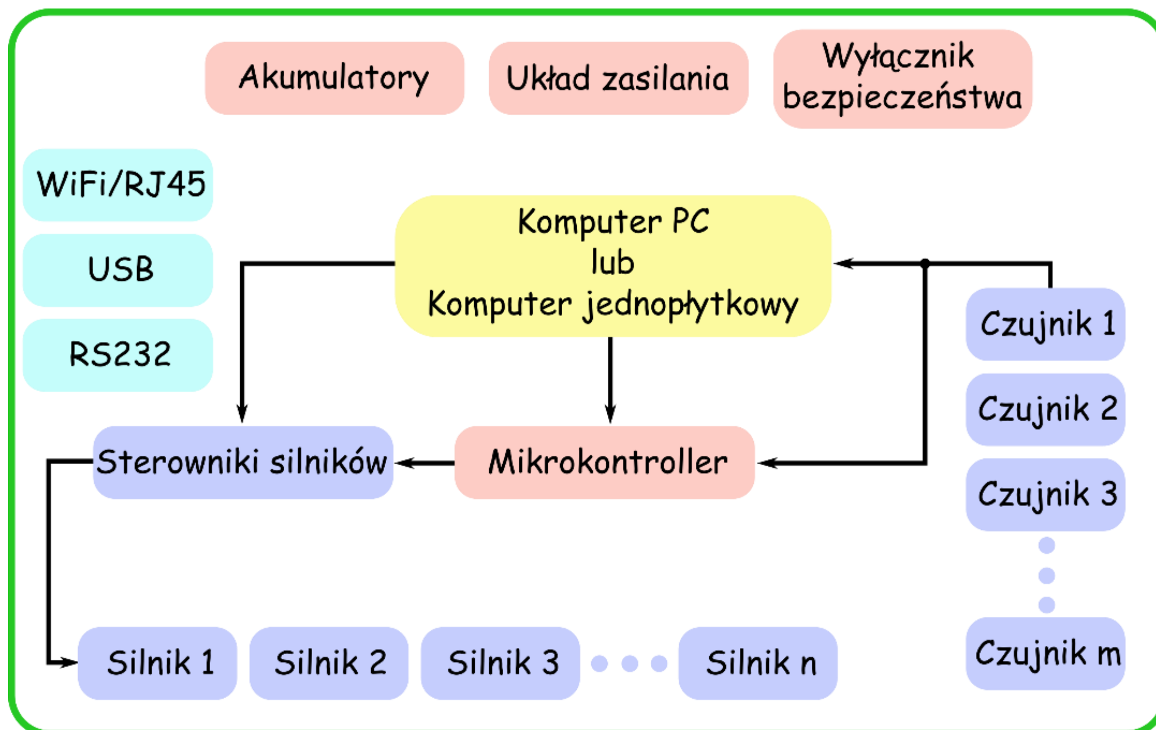
- Czym jest programowanie robotów mobilnych?
- Koncepcja Robot Operating System (ROS)
- Zasada działania ROS
- Tworzenie nowych węzłów
- Kompilacja projektu
- Węzeł publikujący
- Węzeł subskrybujący
- Zmienne globalne
- Środowisko symulacyjne Gazebo
- Pakiet RViz

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Czym jest programowanie robotów mobilnych?

Robot mobilny jest to maszyna z czujnikami, silnikami, siłownikami i jednostką obliczeniową, która pozwala na wykonywanie poleceń zadawanych przez użytkownika lub może podejmować własne decyzje na bazie informacji pozyskanych z czujników. W takim razie możemy powiedzieć, że jednostka obliczeniowa jest „mózgiem” robota. W zależności od złożoności zadania jakie ma robot mobilny realizować może to być:

- mikrokontroler (np. Arduino),
- komputer jednopłytkowy (np. Raspberry PI),
- komputer PC (np. laptop),
- programowalny sterownik logiczny (PLC).

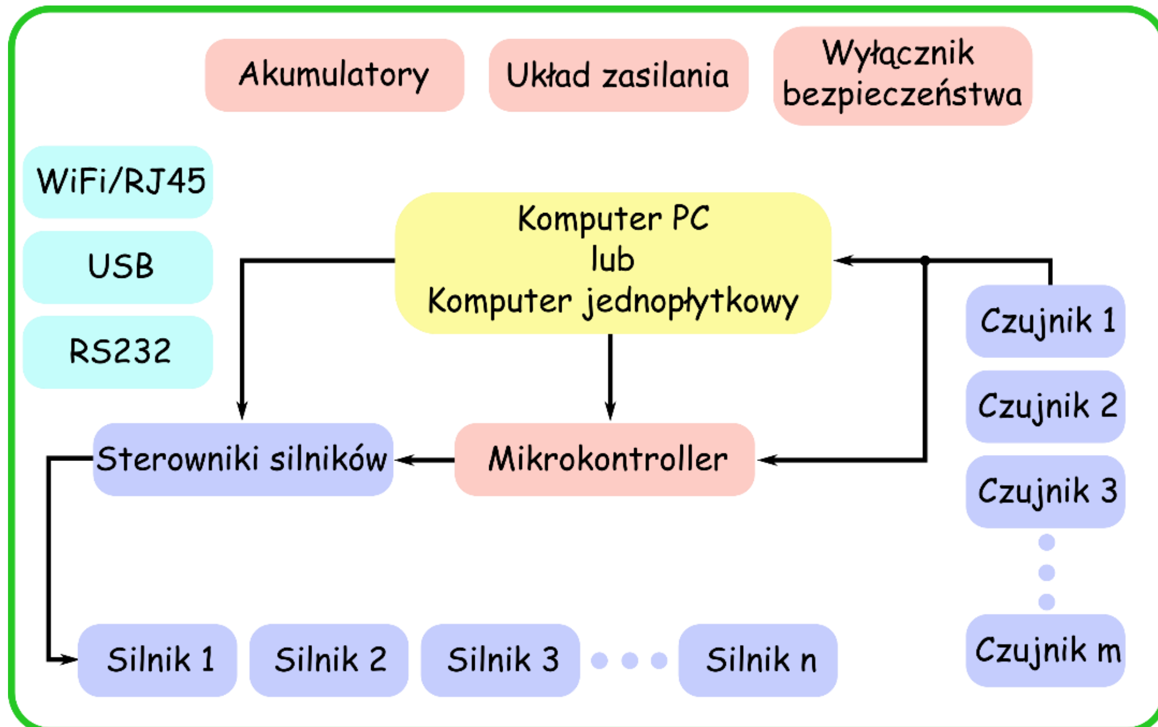


UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Czym jest programowanie robotów mobilnych?

Programowaniem robotów mobilnych nazywamy programowaniem „mózgu" robota. Pisaniem programu na mikrokontroler/PLC, który jest główną jednostką obliczeniową robota i ma realizować specyficzne zadanie wykorzystując dostępne elementy wykonawcze i informacje z czujników nazywamy programowaniem robota. Jednym z najprostszych zadań może być problem podnieś i podaj (tzw. pick-and-place) element z punktu A do B.

W zależności od wykorzystanej głównej jednostki obliczeniowej program może być programowany wykorzystując różne języki programowania: C (mikrokontroler) lub C++, C#, Python (komputer jednopłytkowy/PC) lub schemat drabinkowy i tekst strukturalny (PLC) lub inne specyficzne języki obsługiwane/dedykowane.





UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Istotne cechy z punktu widzenia programowania robota mobilnego:

- Wątki,
- Wysokopoziomowe języki programowania zorientowane obiektowo,
- Sterowanie niskopoziomowe,
- Łatwość prototypowania.,
- Komunikacja miedzyprocesowa,
- Wydajność,
- Wsparcie społeczności,
- Dostępność zewnętrznych bibliotek.

- Wszystkie cechy posiada oprogramowanie nazwane:

Robot Operating System



Robot Operating System (ROS) – podstawowe informacje

- Projekt rozpoczął się na Uniwersytecie Stanforda w 2007 roku,
- Aktualnie jest opracowywany i utrzymywany przez organizację *Open Robotics,*
- Dzięki wbudowanej funkcjonalności pozwala na: wielowątkowość, sterowanie niskopoziomowe, obsługę pakietów, przekazywanie informacji między procesami.
- Obsługuje języki programiwania C/C++ oraz Python,
- W niedalekiej przyszłości obsługiwał będzie również C# oraz Java,
- Integracja z bibliotekami tj. OpenCV,
- Wbudowane implementacja zaawansowanych algorytmów do planowania ścieżki, lokalizacji, mapowania, regulacji itp.,
- Wbudowany symulator Gazebo umożliwiający testowanie opracowywanych algorytmów symulacyjnie,
- Pakiet RViz umożliwiający wizualizację danych ze skanerów laserowych, kamer, mapy, pozycję i orientację robota oraz dane użytkownika.

UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Roboty wspierające ROS:

a) TurtleBot 4 (www.clearpathrobotics.com/turtlebot-4/)
b) Pepper (www.aldebaran.com/en/pepper)
c) Husky (https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/)
d) Universal Robot UR3 (https://www.universal-robots.com/cb3/)



a)



c)



b)



d)

UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Czujniki wspierające ROS:

a) TeraRanger (www.terabee.com)
b) Xsense MTi IMU (www.xsens.com/products/)
c) RPLidar A3 (https://www.slamtec.com/en/Lidar/A3)
d) Intel RealSense (https://realsense.intel.com)



a)



c)



b)



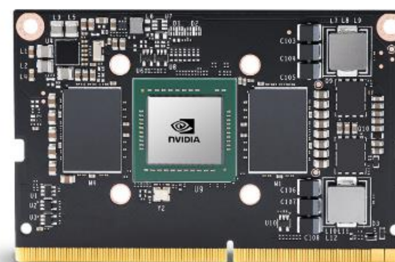
d)

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Komputery jednopłytkowe wspierające ROS:

a) Raspberry Pi 4 (https://www.raspberrypi.com/products/raspberry-pi-4-model-b/)
b) Odroid XU4 (https://www.hardkernel.com/shop/odroid-xu4-special-price/)
c) NVDIA TX2 (https://www.nvidia.com/pl-pl/autonomous-machines/embedded-systems/jetson-tx2/)
d) Intel NUC (www.intel.com/content/www/us/en/products/boards-kits/nuc.html)


a)



c)


b)


d)



UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

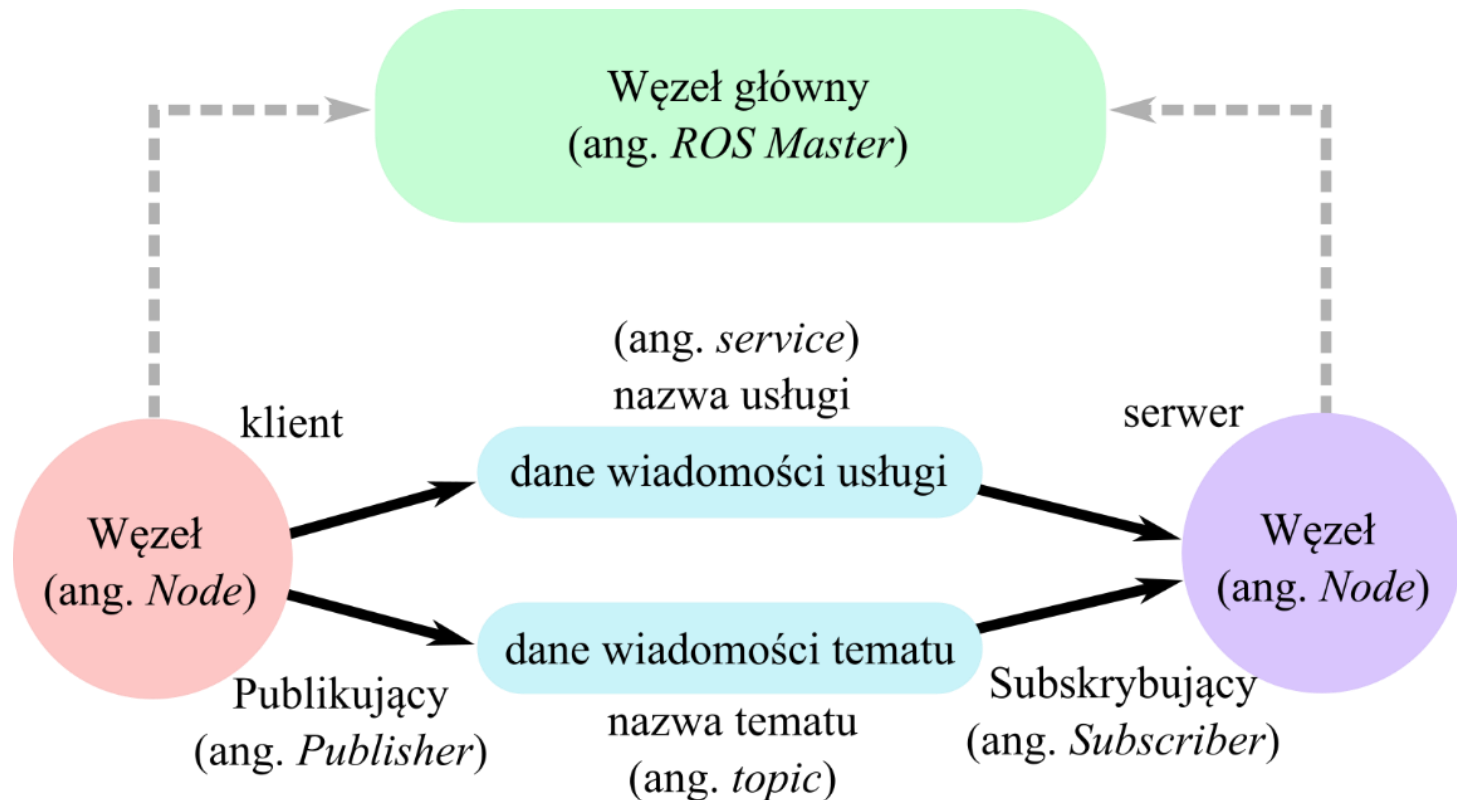
Koncepcja Robot Operating System

W zasadzie to ROS jest szkieletem do komunikacji pomiędzy dwoma programami lub procesami. Każdym pisany przez nas program jest osobnym węzłem (ang. *Node*), który może się komunikować z innymi. Warto zaznaczyć, że nad wszystkim czuwa węzeł główny (ang. *ROS Master*), który jest rdzeniem naszego systemu. Kiedy uruchamiamy nowy węzeł to wysyła on informacje do węzła głównego informując jakie typy danych wysyła i odbiera. Węzeł, który wysyła dane jest nazywany węzłem publikującym (ang. *Publisher*), a węzeł, który odbiera dane to węzeł subskrybujący (ang. *Subscriber*).

Dane wysyłane pomiędzy wątkami mogą być zarówno najprostszymi typami jak liczby całkowite, zmiennoprzecinkowe, łańcuchy znaków itp., ale również złożone typy danych. Wysyłane typy określane są przez *ROS message*. Wiadomości są przekazywane przez łącze zwane *ROS topic*. Każdy temat posiada nazwę oraz typ danych, który przekazuje.

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Koncepcja Robot Operating System





UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Najważniejsze pojęcia związane z Robot Operating System:

- *ROS node (węzeł)*: Proces wykorzystujący interfejs programowania aplikacji ROS,
- *ROS master (węzeł główny)*: Program pośrednik, który łączy węzły
- *ROS parameter server (serwer parametrów)*: Program uruchamiany równocześnie z ROS master. Pozwala na przechowywanie różnych parametrów i wartości.
- *ROS topic (temat)*: Nazwa łącza, przez które węzły mogą przekazywać wiadomości. Każdy węzeł może subskrybować lub publikować dowolną liczbę tematów.
- *ROS message (wiadomość)*: Wiadomości są przekazywane przez tematy. Istnieją wbudowane typy wiadomości, jak również użytkownik może stworzyć własną.
- *ROS service (usługa)*: Usługi pozwalają na zastosowanie mechanizmu pytanie-odpowiedź. Usługa wywołuje funkcje węzła dostarczającego usługę (serwer), która może zostać wywołana przez inny węzeł (klienta) w dowolnym momencie.
- *ROS bag (worek)*: Metoda pozwalająca zapisać i odtworzyć zapisane tematy. Również wykorzystywana do logowania danych robota w celu późniejszego ich wykorzystania.



Główne komendy konsolowe Robot Operating System:

- *roscore:* najwazniejsza komenda w ROS – uruchamia *ROS master, ROS parameter server* oraz *logging node (węzeł logujący)*.
- *rosnode:* funkcja pozwalająca na zarządzanie uruchomionymi węzłami. Na przykład wypisanie listy uruchomionych węzłów: *rosnode list*.
- *rostopic:* dostarcza informacje o tematach publikowanych i subskrybowanych. Pozwala na wypisanie dostępnych tematów *rostopic list*, podsłuchanie przesyłanych danych w danym temacie *rostopic echo /nazwa_tematu*, lub opublikowanie danych w danym temacie *rostopic pub nazwa_tematu typ_wiadomości dane*.
- *rosparam:* pozwala na wypisanie i edytowanie zapisanych zmiennych w serwerze parametrów (*ROS parameter server*). Wypisanie: *rosparam list*, zmiana wartości: *rosparam set nazwa_zmiennej wartość_zmiennej,* odczyt wartości *rosparam get nazwa_zmiennej.*
- *rosrun:* uruchamia węzeł: *rosrun nazwa_pakietu_ros nazwa_węzła.*
- *roslaunch:* przydatka komenda ROS pozwalająca na uruchomienie jednoczesne kilku węzłów. Komenda *roslaunch nazwa_pakietu_ros nazwa_pliku_launch* pozwala na uruchomienie wszystkich węzłów oraz niezbędnych węzłów zależnych zdefiniowanych w pliku XML **.launch*. Automatycznie uruchamiania ROS master.



UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Przykładowe węzły: *talker* i *listener*

Przejdźmy teraz do prostego przykładu, gdzie uruchomimy węzeł główny (*ROS master*), przykładowy węzeł publikujący (*talker*) i węzeł subskrybujący (*listener*). Tematem jaki będzie publikowany to */chatter*. Uruchamiając trzy konsole (można wykorzystać program *Terminator*) należy w każdej z nich uruchomić kolejno:

Węzeł główny:

```
$  roscore
```

węzeł publikujący temat */chatter*:

```
$  rosrun roscpp_tutorials talker
```

węzeł subkrybujący temat */chatter*:

```
$  rosrun roscpp_tutorials listener
```



Przykładowe węzły: *talker* i *listener*

ros@ros-VirtualBox: ~

roscore http://ros-VirtualBox:11311/ 132x9

```
auto-starting new master
process[master]: started with pid [3090]
ROS_MASTER_URI=http://ros-VirtualBox:11311/

setting /run_id to c4d796ee-aaa3-11ed-8dea-7f04338e31ed
process[rosout-1]: started with pid [3100]
started core service [/rosout]
```

ros@ros-VirtualBox: ~ 132x11

```
ros@ros-VirtualBox:~$ rosrun roscpp_tutorials talker
[ INFO] [1676185646.463091810]: hello world 0
[ INFO] [1676185646.563457835]: hello world 1
[ INFO] [1676185646.672944123]: hello world 2
[ INFO] [1676185646.763302746]: hello world 3
[ INFO] [1676185646.864816020]: hello world 4
[ INFO] [1676185646.964616172]: hello world 5
[ INFO] [1676185647.065436275]: hello world 6
[ INFO] [1676185647.190838244]: hello world 7
[ INFO] [1676185647.272571957]: hello world 8
[ INFO] [1676185647.373560142]: hello world 9
```

ros@ros-VirtualBox: ~ 132x11

```
ros@ros-VirtualBox:~$ rosrun roscpp_tutorials listener
[ INFO] [1676185646.764065436]: I heard: [hello world 3]
[ INFO] [1676185646.865175073]: I heard: [hello world 4]
[ INFO] [1676185646.964953339]: I heard: [hello world 5]
[ INFO] [1676185647.065688615]: I heard: [hello world 6]
[ INFO] [1676185647.191713097]: I heard: [hello world 7]
[ INFO] [1676185647.273196611]: I heard: [hello world 8]
[ INFO] [1676185647.373953479]: I heard: [hello world 9]
[ INFO] [1676185647.472972171]: I heard: [hello world 10]
[ INFO] [1676185647.564075730]: I heard: [hello world 11]
[ INFO] [1676185647.663669711]: I heard: [hello world 12]
[ INFO] [1676185647.777128839]: I heard: [hello world 13]
```



UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Przykładowe węzły: *talker* i *listener* (plik .launch)

Jak wspomniane było wcześniej, istnieje komenda umożliwiająca uruchomienie węzła głównego i wielu wątków równocześnie. Powyższy przykład możemy uruchomić jedną komendą wykorzystując przykładowy plik *talker_listener.launch:*

```
$ roslaunch roscpp_tutorials talker_listener.launch
```

ros@ros-VirtualBox: ~

ros@ros-VirtualBox: ~ 132x35

```
setting /run_id to ba335650-aaa4-11ed-8dea-7f04338e31ed
process[rosout-1]: started with pid [3350]
started core service [/rosout]
process[listener-2]: started with pid [3354]
process[talker-3]: started with pid [3357]
[ INFO] [1676185999.858735153]: hello world 0
[ INFO] [1676185999.959676793]: hello world 1
[ INFO] [1676186000.059456222]: hello world 2
[ INFO] [1676186000.167748180]: hello world 3
[ INFO] [1676186000.168095765]: I heard: [hello world 3]
[ INFO] [1676186000.265100322]: hello world 4
[ INFO] [1676186000.265667481]: I heard: [hello world 4]
[ INFO] [1676186000.358992086]: hello world 5
[ INFO] [1676186000.359489026]: I heard: [hello world 5]
[ INFO] [1676186000.459412445]: hello world 6
[ INFO] [1676186000.459996811]: I heard: [hello world 6]
[ INFO] [1676186000.573111577]: hello world 7
[ INFO] [1676186000.573828269]: I heard: [hello world 7]
```



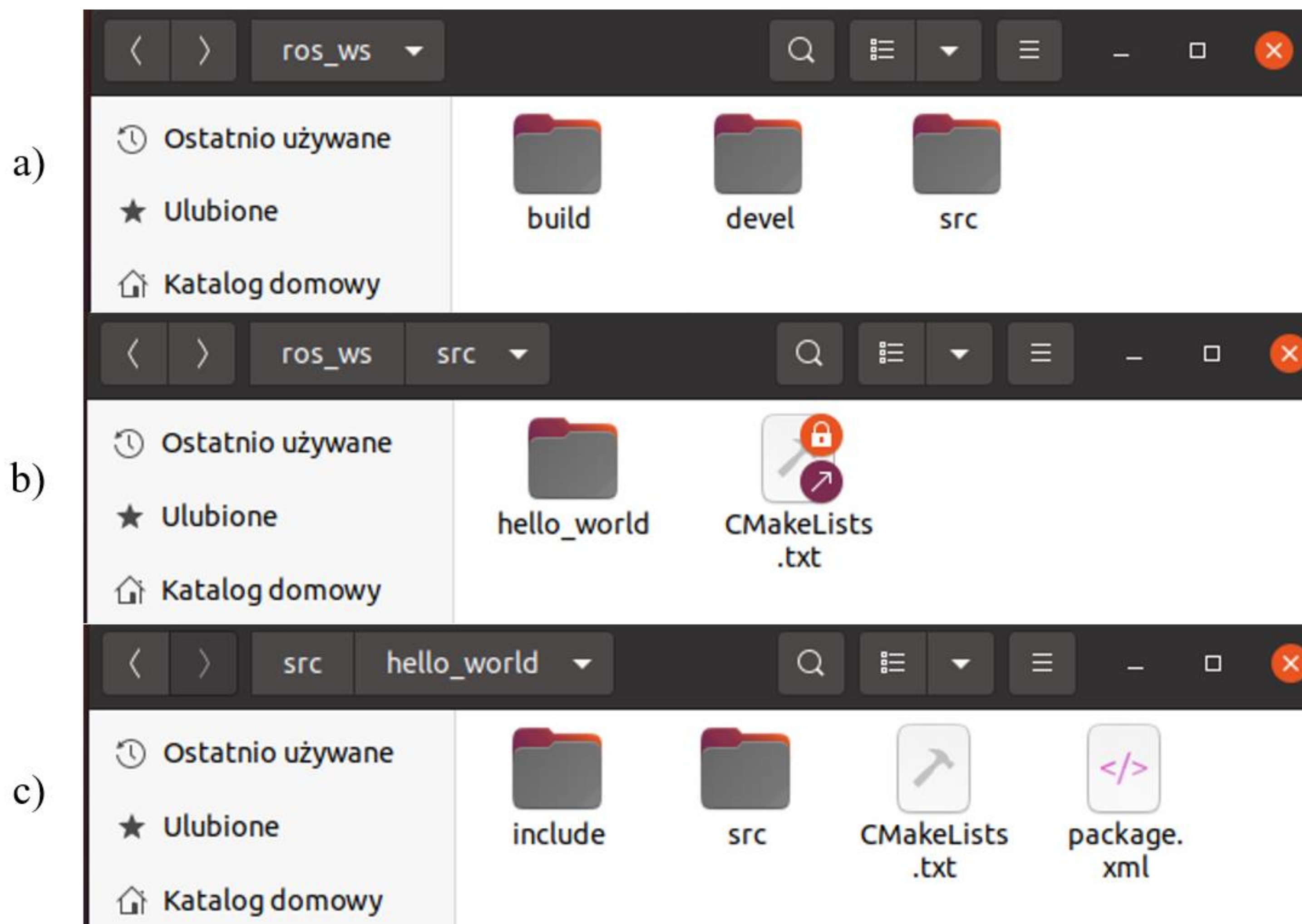
Struktura plików

W celu stworzenia wpłąsnych węzłów musimy stworzyć przestrzeń roboczą dla ROS i następnie naszego pakietu, który będziemy implementować. W tym celu wykonujemy następujące komendy:

```
$  mkdir -p ~/ros_ws/src
$  cd ros_ws/src
$  catkin_init_workspace
$  cd ~/ros_ws
$  catkin_make
$  echo "source ~/ros_ws/deve/setup.bash" >> ~/.bashrc
$  cd ~/ros_ws/src
$  catkin_create_pkg hello_world roscpp rospy std_msg
```

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Struktura plików





Struktura plików pakietu *Hello_world:*

- *CMakeLists.txt* – lista komend do zbudowania pakietu i stworzenia plików wykonywalnych,
- *package.xml* – zawiera informacje o zależnościach i inne informacje dotyczące pakietu,
- *src* – katalog źródłowy na pliki źródłowe C++,
- *include* – katalog dla plików nagłówkowych C++ i bibliotek zewnętrznych.
- *scripts* – katalog źródłowy na pliki źródłowe Python,
- *msg* – katalog na nowe typy wiadomości, które zdefiniowaliśmy na potrzeby pakietu. Warto dodać, że są one niezależne od wykorzystywanego języka programowania,
- *srv* – katalog na pliki opisujące serwisy, w których można wyróżnić zapytania (ang. *request*) i odpowiedzi (ang. *response*),
- *launch* – katalog na pliki *launch*.



Dodanie węzła do plików wykonywalnych:

W celu dodania węzła zaimplementowanego w jezyku C++ należy dodać następujący wiersz w pliku *CMakeLists.txt:*

```
add_executable(nazwa_węzła nazwa_pliku.cpp)
```

gdzie pierwszy argument jest nazwą widoczną z poziomu ROS, a drugi to nazwa pliku wykonywalnego.

W przypadku węzłą zaimplementowanego w języku Python wystarczy dodać uprawnienia wykonywania stworzonego pliku następującą komendą:

```
$  chmod +x ~/ros_ws/src/hello_world/scripts/nazwa_pliku.py
```

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Tworzenie węzłów – Importowanie bibliotek

Pierwszą czynnością podczas implementacji nowych węzłów musi być zaimportowanie bibliotek podstawowych ROS:

C++	Python
`#include "ros/ros.h"`	`import rospy`

Importowanie typów wiadomości publikowanych przez węzły publikujące wykonuje się następująco:

C++	Python
`#include "std_msgs/String.h` `#include "std_msgs/Int32.h`	`from std_msgs.msg import String` `from std_msgs.msg import Int32`



UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Tworzenie węzłów – Inicjalizacja węzła

Poleceniem wykonanym w pierwszej kolejności powinna być rejestracja węzła w systemie poprzez następującą komendę:

C++	Python
`ros::init(argc, argv, "nazwa węzła");` `ros::NodeHandle nh;`	`rospy.init_node('nazwa_węzła’, anonymous=True)`

Uchwyt do węzła (ang. *node handle*) w języku C++ należy jawnie zdefiniować, natomiast w przypadku języka Python, uchwyt jest zdefiniowany wewnątrz modułu *rospy.*



Tworzenie węzłów – węzeł publikujący

Deklarację, że nasz węzeł będzie publikował wiadomości w ramach tematu wykonuje się w następujący sposób:

C++

```
ros::Publisher pub = nh.advertise<typ_wiadomosci_ROS>("nazwa_tematu", 10);
```

Python

```
pub = rospy.Publisher(“nazwa_tematu”, typ_wiadomosci_ROS, queue_size=10)
```

Publikowanie wiadomości realizuje się w następujący sposób:

C++

```
std_msgs::String msg;
msg.data = "String data"
ros::Publisher chatter_pub = nh.advertise<std_msgs::String>("chatter", 10);
chatter_pub.publish(msg);
```

Python

```
msg = String()
msg.data = "string data"
pub = rospy.Publisher('chatter', String, queue_size=10)
pub.publish(msg)
```



Tworzenie węzłów – węzeł subskrybujący

Deklarację, że nasz węzeł będzie nasłuchiwał wiadomości w ramach tematu wykonuje się w następujący sposób:

C++

```
ros::Subscriber subscriber_obj = nh.subscribe("nazwa_tematu", 10, callback)
```

Python

```
rospy.Subscriber("nazwa_tematu ", typ_wiadomosci_ROS, callback)
```

W momencie opublikowania nowej wiadomości w subskrybowanym temacie, zostanie wywołana zadeklarowane funkcją *callback*, którą definiujemy w następujący sposób:

C++

```
void callback(const ros_message_const_pointer &pointer) {
   // pobieranie danych: pointer->data
}
```

Python

```
def callback(data):
   # pobieranie danych: data.data
```

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Tworzenie węzłów – serwer parametrów

Odczyt zmiennych globalnych z serwera parametrów (*ROS parameter server*) wykonujemy w następujący sposób:

C++

```
std::string nazwa_zmienne_globalnej;
if (nh.getParam("/nazwa_zmienne_globalnej ", nazwa_zmienne_globalnej)) {
   // ...
}
```

Python

```
nazwa_zmienne_globalnej = rospy.get_param("/nazwa_zmienne_globalnej ")
```

Zapis do serwera parametrów:

C++

```
nh.setParam("/nazwa_zmienne_globalnej ", „Hello World!”);
```

Python

```
rospy.set_param('~private_int', '2')
```



UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Tworzenie węzłów – synchronizacja węzła z ROS

[C++] Ostatnim zagadnieniem jest funkcja *ros::spinOnce()* oraz *ros::spin()*. Musimy wykonać tę funkcję w celu wyzwolenia procesu prośby o subskrypcję oraz publikowanie tematu. Używamy *spinOnce()* po publikowaniu tematu, a *spin()* jeżeli tylko subskrybujemy temat. W przypadku obu czynności powinniśmy zastosować *spinOnce()*.

[Python] Analogicznie do C++, *rospy.spin()* w przypadku publikowania, *rospy.sleep()* w przypadku subskrybowania lub obu operacji.

C++

```
ros::Rate r(10); // Inicjalizacja pętli na częstotliwość 10 Hz
while(ros::ok()) {
   // ...
   ros::spinOnce();
   r.sleep(); // Wywoływane w głównej pętli programu
}
```

Python

```
rate = rospy.Rate(10)  # Inicjalizacja pętli na częstotliwość 10 Hz
while not rospy.is_shutdown():
   # ...
   rate.sleep() # Wywoływane w głównej pętli programu
```

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Tworzenie węzłów – wyświetlanie informacji w konsoli węzła

Mamy kilka rodzajów statusów, które możemy wyświetlić:

- Status informacyjny:

C++	Python
`ROS_INFO(string_msg, args);`	`rospy.loginfo(msg, *args)`

- Status ostrzegawczy:

`ROS_WARN(string_msg, args);`	`rospy.logwarn(msg, *args)`

- Status debugowania:

`ROS_DEBUG(string_msg, args);`	`rospy.logdebug(msg, *args)`

- Status błędu:

`ROS_ERROR(string_msg, args);`	`rospy.logerr(msg, *args)`

- Status błędu fatalnego:

`ROS_FATAL(string_msg, args);`	`rospy.logfatal(msg, *args)`



UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Tworzenie pliku *.launch*

Napisane węzły możemy uruchomić wykorzystując znaną komendę *rosrun*. Jak pokazywałem wcześniej, można stworzyć plik uruchomieniowy, który uruchomi wiele węzłów równocześnie. Sam plik *launch* umożliwia uruchamianie węzłów z wieloma opcjami, tj. automatyczne uruchamianie w przypadku zakończenia jego procesu, przekazywanie parametrów wejściowych. Dodatkowo możemy tworzyć zmienne dla serwera parametrów, Poniżej przykład:

```
<launch>
   <!– Parametr liczby zmiennoprzecinkowej dla serwera parametrów -->
   <param name="my_double" value="3.14159" type="double" />
   <!– Parametr liczby całkowitej dla serwera parametrów -->
   <param name="my_int" value="3" type="int" />
   <!– Parametr łańcucha znaków dla serwera parametrów -->
   <param name="my_str" value="Hello" type="str" />
   <!– Parametr logiczny dla serwera parametrów -->
   <param name="my_PI" value="false" type="bool" />
   <!– Uruchomienie węzła zaimplementowanego w C++ -->
   <node name="listener_node" pkg="hello_world" type="listener" output="screen"/>
   <!– Uruchomienie węzła zaimplementowanego w Python -->
   <node name="talker_node" pkg="hello_world" type="talker.py" output="screen"/>
</launch>
```

W celu uruchomienia pliku *launch* należy nadać mu uprawnienia do wykonywania poprzez komendę w konsoli:

```
$ sudo chmod +x talker_listener.launch
```



UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Gazebo – środowisko symulacyjne

Gazebo jest otwartym oprogramowaniem. Jest to symulator robotów w 3D z symulowaną fizyką rzeczywistego świata o wysokiej wierności. Pozwala to na szybkie prototypowanie i testowanie algorytmów, jak również projektowanie robotów w cyfrowym środowisku.

Zacznijmy od uruchomienia przykładowego świata (ang. *world*) z robotem TurtleBot3 Waffle PI:

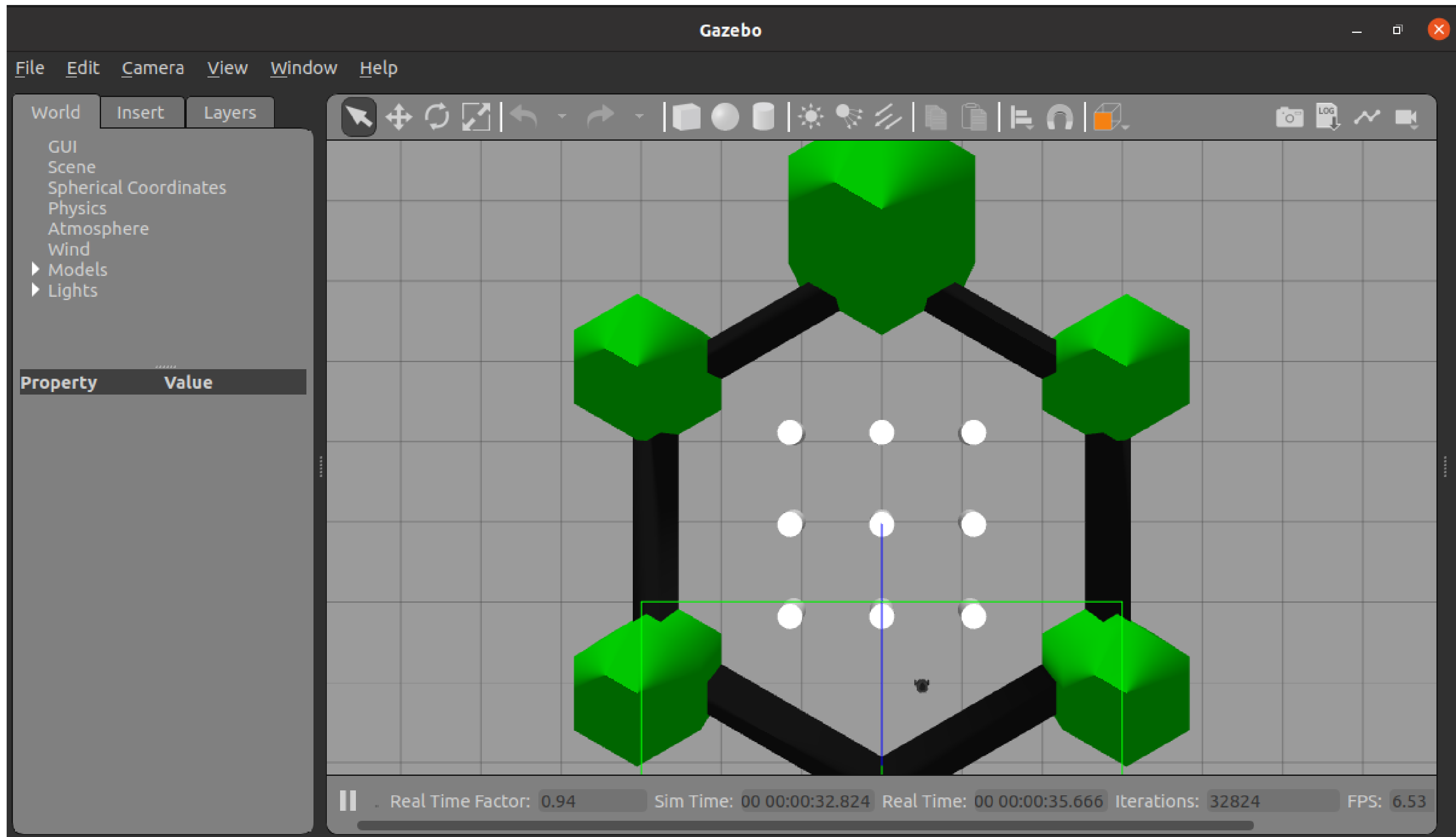
```
$  export TURTLEBOT3_MODEL=waffle_pi
$  roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Warto dodać definicję modelu do pliku .bashrc, żebyśmy w przyszłości nie musieli go definiować za każdym razem, gdy uruchomimy nową konsolę:

```
$  echo “export TURTLEBOT3_MODEL=waffle_pi” >> ~/.bashrc
```

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Gazebo – środowisko symulacyjne





RViz – wizualizacja danych

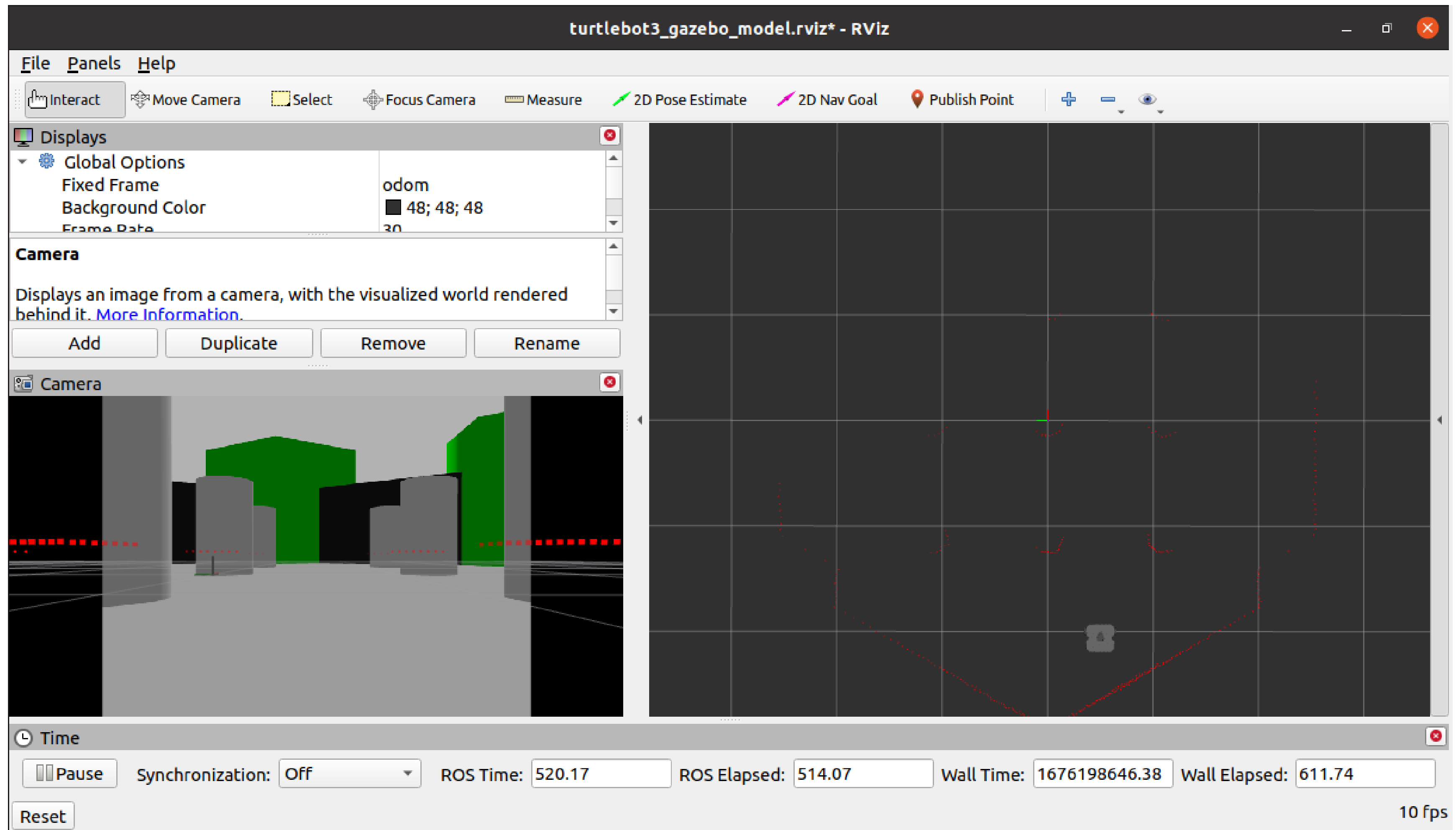
Rviz umożliwia wizualizację danych ze skanera laserowego 360 stopni, kamery RGB, a ponadto umożliwia wizualizację swoich własnych danych, które możemy wysyłać z naszych własnych węzłów. Dzięki temu prototypowania algorytmów jest dużo prostsze, ponieważ możemy w graficznej formie przedstawić niektóre elementy naszego algorytmu.

W celu uruchomienia RViz skonfigurowanego specjalnie dla robota TurtleBot 3 należy wykonać następującą komendę (nie wyłączając środowiska Gazebo):

```
$  roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch
```

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

RViz – wizualizacja danych



UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Gazebo i RViz – podstawowe operacje myszką

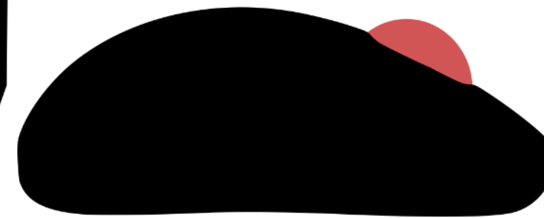
Gazebo

Lewy Przycisk Myszy:
Przytrzymaj = przesuń
Shift + przytrzymaj = obróć
Podwójne kliknięcie = przejdź do punktu



Prawy Przycisk Myszy:
Przytrzymaj i przesuń w górę = przybliż
Przytrzymaj i przesuń w dół = oddal
Pojedyncze kliknięcie = otwórz menu

Rolka (ang. *scroll*) = Przybliż/oddal



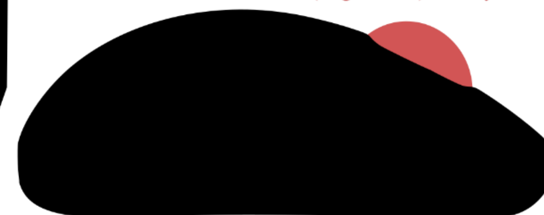
RViz

Lewy Przycisk Myszy:
Przytrzymaj = obróć
Shift + przytrzymaj = przesuń



Prawy Przycisk Myszy:
Przytrzymaj i przesuń w górę = przybliż
Przytrzymaj i przesuń w dół = oddal

Rolka (ang. *scroll*) = Przybliż/oddal





Gazebo i Rviz – bezkolizyjny ruch robota TurtleBot 3

W tym celu należy zatrzymać węzeł odpowiadający za ręczne sterowanie i wykonać komendę:

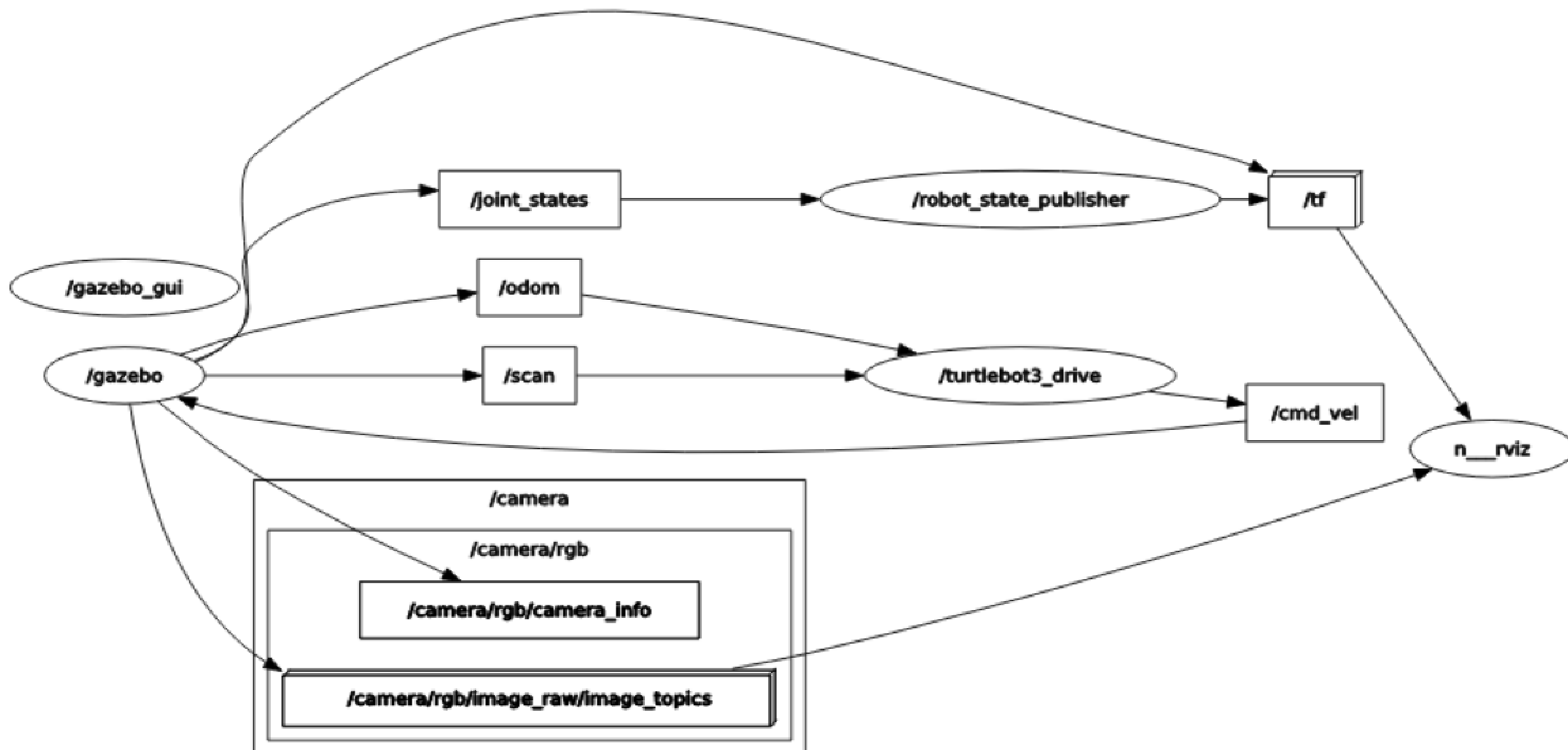
```
$  roslaunch turtlebot3_gazebo turtlebot3_simulation.launch
```

Przydatnym polecenime jest *rqt_graph*. Rysuje on zależności pomiędzy wszystkimi węzłami, tematami i serwisami:

```
$  rqt_graph
```

UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Gazebo i Rviz – bezkolizyjny ruch robota TurtleBot 3





UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Gazebo i Rviz – mapowanie terenu

Przejdziemy do prezentacji przykładowej aplikacji SLAM (równoczesna lokalizacja i mapowanie). W tym celu musimy zainstalować pakiet gmapping (chyba, że już to zrobiliśmy):

```
$  sudo apt install ros-noetic-slam-gmapping
```

A następnie uruchomić przykładową aplikację wywołując polecenie:

```
$  roslaunch turtlebot3_gazebo turtlebot3_world.launch
$  roslaunch turtlebot3_slam turtlebot3_slam.launch
   slam_methods:=gmapping
```

Teraz mamy dwie możliwości: możemy uruchomić ręczne sterowanie robotem:

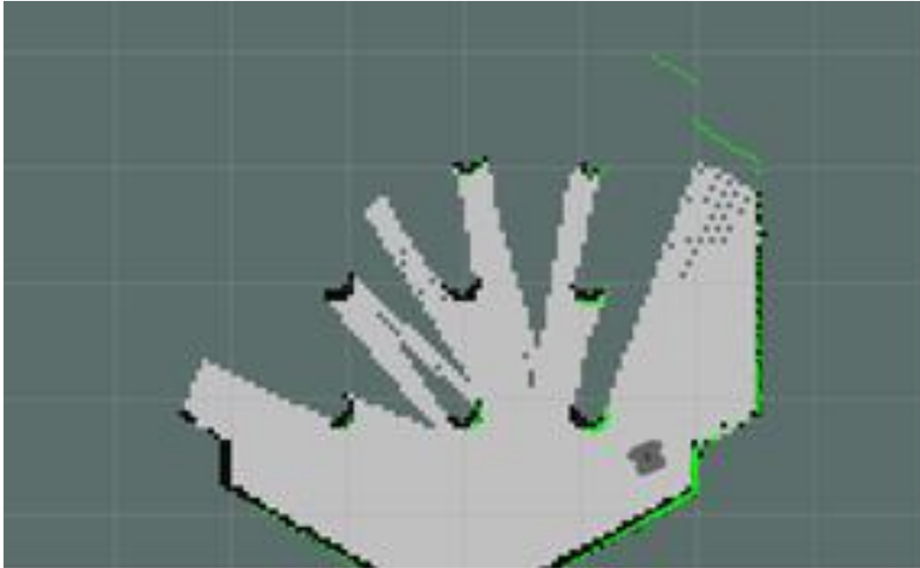
```
$  roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

lub autonomiczny i bezkolizyjny ruch z wcześniejszego przykładu. Osobiście uruchomiłem autonomiczny ruch poleceniem:

```
$  roslaunch turtlebot3_gazebo turtlebot3_simulation.launch
```

UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU
Wydział Fizyki, Astronomii
i Informatyki Stosowanej

Gazebo i Rviz – mapowanie terenu





Po zakończeniu mapowania terenu możemy zapisać zbudowaną mapę wykonując polecenie:

```
$  rosrun map_server map_saver -f ~/map
```

W ten sposób zapisaliśmy mapę do pliku *map.pgm*. Jest to plik o rozszerzeniu *.pgm* (ang. *portable gray map*). To co warto zaznaczyć to fakt, że ten plik możemy w prosty sposób edytować np. w Paint, żeby usunąć nieprawidłowości lub błędy wynikające z „tymczasowych" przeszkód, których na mapie nie chcemy mieć.



Gazebo i Rviz – Algorytmy lokalizacji i planowania ścieżki

Skoro mamy mapę, to teraz możemy ją wykorzystać do lokalizacji naszego robota. Ponownie wyłączmy całą symulację i uruchomimy ją ponownie wczytując wcześniej zbudowaną mapę, ale najpierw należy się upewnić, ze mamy zainstalowany pakiet związany z algorytmem planowania ścieżki dynamicznego okna (ang. *Dynamic Widnow Appraoch*, DWA):

```
$   sudo apt-get install ros-kinetic-dwa-local-planner
```

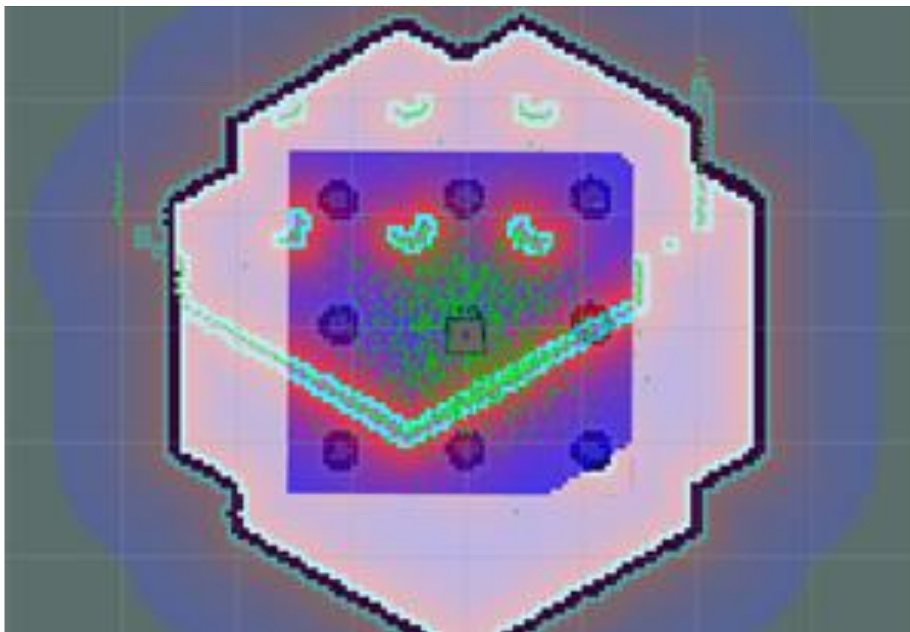
Następnie uruchamiamy środowisko:

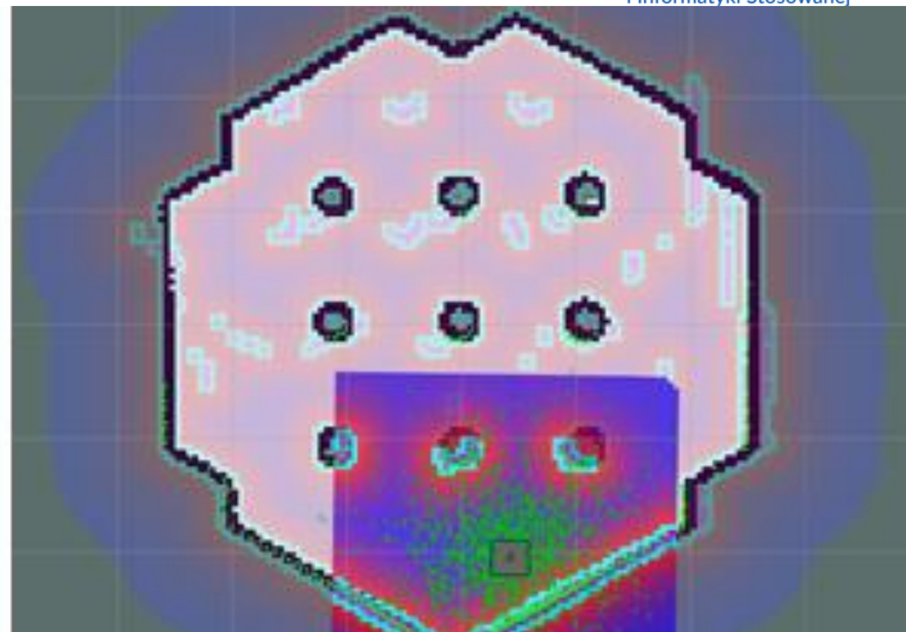
```
$   roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

```
$   roslaunch turtlebot3_navigation turtlebot3_navigation.launch
map_file:=$HOME/map.yaml
```

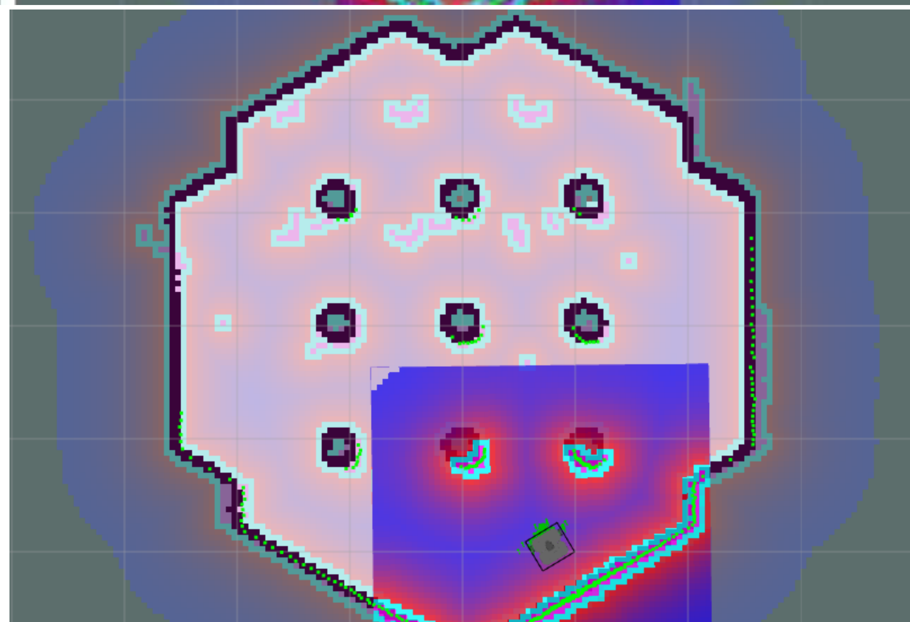
Po uruchomieniu AMCL (ang. *Adaptive Monte Carlo Localization approach*, adaptacyjny algorytm lokalizacji Monte Carlo) konieczne jest ręczne ustawienie początkowej pozycji naszego robota względem mapy (patrz lewą kolumnę Rys. 6 8). W tym celu wybieramy z paska narzędzi w RViz opcję *2D pose estimate*. i wybieramy pozycję i orientację robota na mapie tak, żeby aktualny odczyt ze skanera laserowego pokrywał się z rzeczywistą mapą.

UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej





- Wczytana mapa musi zostać skorygowana do początkowej pozycji
- Po wykonaniu kilku ruchów robotem w trybie manualny, algorytm AMCL z dużą dokładnością estymuje pozycję rzeczywistą robota



UNIWERSYTET MIKOŁAJA KOPERNIKA W TORUNIU
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Gazebo i Rviz – Algorytmy lokalizacji i planowania ścieżki

